Effective API Navigation and Reuse

Awny Alnusair, Tian Zhao
Department of Computer Science
University of Wisconsin-Milwaukee, USA
{*alnusair, tzhao*}*@uwm.edu*

Eric Bodden
Department of Computer Science
Technische Universität Darmstadt, Germany
bodden@acm.org

Abstract

Most reuse libraries come with few source-code examples that demonstrate how the library at hand should be used. We have developed a source-code recommendation approach for constructing and delivering relevant code snippets that programmers can use to complete a certain programming task. Our approach is semantic-based; relying on an explicit ontological representation of source-code. We argue that such representation opens new doors for an improved recommendation mechanism that ensures relevancy and accuracy. Current recommendation systems require an existing repository of relevant code samples. However, for many libraries, such a repository does not exist. Therefore, we instead utilize points-to analysis to infer precise type information of library components. We have backed our approach with a tool that has been tested on multiple libraries. The obtained results are promising and demonstrate the effectiveness of our approach.

Keywords: Code reuse, Ontology, Points-to analysis

1. Introduction

Programmers often reuse code in many ways. One common way is to access libraries of reusable components, or to plug into application frameworks. Unfortunately, many libraries and frameworks are not intuitive to use. While some exceptional pieces of software may be documented well, it is often the case that libraries lack informative API documentation, and lack sufficient source-code examples that would explain a particular library feature. When such an example does exist, it can be very helpful: programmers can often simply copy the example into the current project and then adapt it to the new context, thus enabling a particular API programming task to be completed rather quickly. In this paper, we present an approach for automatic source-code recommendation. Our approach is based on the idea that many programming tasks require programmers to compose a chain of method calls that convert a given source object of some particular type to a target object of some other type. Thus, the programmer needs to answer object-instantiation queries of the form (*Source object* ⇒*Destination object*). In the special case where the *Source* object is not specified, the object-instantiation problem is reduced to either a simple constructor invocation or a static method invocation. For illustration, consider a programmer trying to reuse Jena[1]; an open-source Java framework for building Semantic Web applications. At some point, the programmer wishes to programmatically construct a fragment of a *model* based on a template in a given semantic query. She would start with a `Query` object obtained from a `String` representation of the query and wishes to end up with a `Model` object representing the newly constructed model. This programming task can be seen as an object-instantiation task of the form (*Query* ⇒*Model*). The following code snippet outlines a sample solution for this task:

```
Query query =
   QueryFactory.create(queryString);
QueryExecution qe =
   QueryExecutionFactory.create(query);
Model m = qe.executeConstruct();
```

For a developer who is unfamiliar with Jena, accomplishing this task may not be easy. In particular, identifying the proper call chains and the various static method invocations require the programmer to have a substantial knowledge of the framework's structure. Even worse, many frameworks do not provide type-specific methods that may free developers from using downward type casts. This complicates the process of composing code snippets, as casts must be inserted to make the snippets compile. Our proposal to automatic code recommendation tackles these issues effectively by "ontologizing" source-code knowledge.

An ontology is an explicit specification of a conceptualization [4]. It provides means to formally describe concepts, objects, properties and other entities in a domain of discourse, and to describe the relationships that hold among

[1] `http://jena.sourceforge.net/`

these concepts. We thus use ontology formalisms to represent software assets by building a knowledge base that is automatically populated with instances representing source-code artifacts. Our approach uses this knowledge base to identify and retrieve relevant code snippets.

Besides addressing knowledge representation issues, our approach improves on the existing state-of-the-art in the following ways: *a*) Unlike other recommendation approaches, we neither require a repository of sample code to mine for snippets, nor do we require our tool to be backed by a source-code search engine to obtain these samples. These requirements have been identified as one of the major limitations of current recommendation systems [10]; *b*) Since the structure of an API usually contains too little information to obtain useful code snippets that require special features (e.g. determining the legality of a type cast), we use inter-procedural points-to analysis to enrich our knowledge base with information about the possible runtime behavior of the API; *c*) We provide a context-sensitive approach that analyzes the user's code when constructing, ranking, and delivering code-snippet candidates; and *d*) Similar to other approaches, we traverse a graph representation of source-code to find a path between the source and the destination objects. However, since our graph is based on an ontology model, it is enriched with additional data that guides the search and helps us obtain more precise results.

2. Related work

Researchers have proposed many code recommendation techniques; all of which tackle the problem from different perspectives. Data Mining-based techniques try to reveal usage patterns of program components from a corpus of existing code examples. This is usually accomplished by extracting association rules which incorporate taxonomies of inheritance relationships [8], or by applying frequent-sequence mining and clustering techniques [15] to extract API methods that are frequently invoked in sequence. Such data mining techniques often suffer from scalability and rule-complexity issues. Traditional Information Retrieval techniques, on the other hand, circumvent some of these complexity issues. They allow users to formulate keyword queries to retrieve source-code samples ranked based on the match between the query and the obtained name-based indices [11] or latent semantic-based indices [9]. Due to the nature of the schemes and the keyword-based search employed, traditional keyword-based recommendation systems are usually very imprecise.

Some other approaches do in fact recommend personalized code snippets when queried. These approaches base their recommendation on analyzing a large corpus of sample client code collected using Google Code Search (GCS) (PARSEWeb [13]), or by searching in a pre-populated local repository (Strathcona [5], Prospector [7], and XSnippet [12]). Strathcona for example, is a recommendation tool that uses heuristics to match the structure of the code under development (structural context) to the structure of the code in a source-code repository. PARSEWeb, Prospector, and XSnippet, on the other hand, are more focused on answering specific object-instantiation queries.

Although these approaches take important steps in the right direction, we believe that there are fundamental issues related to the mechanisms used for data processing and data representation. Firstly, with the exception of Prospector, other tools rely on a hard-to-find repository populated with client code that expresses *good* usages of the framework. Prospector, however, does analyze API signatures for the most part, but still relies on a repository to handle special features such as downcasts. For tools that collect code samples from the web, the obtained samples are usually incomplete fragments that cannot be analyzed precisely. Secondly, traditional knowledge-representation mechanisms and hard-coded heuristics affect the quality of the retrieved results and in most cases are highly unoptimized. None of these approaches uses a formal and explicit representation of either the user context or the source-code structure. Whether the representation mechanism used is a relational database (Strathcona) or traditional graph-based representation (PARSEWeb, XSnippet and Prospector), we hypothesize that encoding the representation using ontology formalisms improves the search for relevant code snippets. Ontologies can naturally combine knowledge from multiple sources (contexts) and then allow for computing entailments from this combined knowledge.

3. Ontology-based code recommendation

An ontology is an explicit data model for a particular domain. It consists of machine-interpretable definitions of classes that formally describe domain concepts, relationships between classes and their structural properties; and constraints expressed as axioms. Therefore, we use our own Source Code Representation Ontology (SCRO) to formally represent the conceptual source-code knowledge of the user context as well as software libraries used in the user's project. SCRO provides a base model for understanding the relationships and dependencies among source-code artifacts. It captures major concepts and features of object-oriented programs including encapsulation, inheritance, method overloading, method overriding, and method signature information. SCRO's knowledge is represented using the OWL-DL[2] ontology language. OWL is a web-based language used for capturing relationship semantics among domain concepts, OWL-DL is a subset of OWL

[2] http://www.w3.org/TR/owl-guide/

based on Description Logic and has desirable properties for reasoning systems.

SCRO defines various OWL classes and subclasses. These classes map directly to code elements and collectively represent the most important concepts found in object-oriented programs. Furthermore, we define various object properties, sub-properties, and ontological axioms within SCRO to represent various relationships among ontological concepts. SCRO is precise, well documented, and available online [1] so it can be reused by other semantic-based applications that require source-code knowledge.

After having created the ontology structure, one next needs to populate the knowledge base with ontological instances (OWL individuals) that represent various concepts in the ontology. In the context of source-code recommendation, we need to populate SCRO with instances from various sources. As the main source of information, we consider the framework(s) currently being reused. Furthermore, to rank the retrieved snippets, we also take into account the user context depicted by the current project under development.

To that end, we have built a knowledge generator for Java. The generated semantic instances are serialized using the RDF[3] language. RDF is suitable for describing resources and provides a data model for representing machine-processable semantics of data. For each framework parsed, our knowledge generator builds an RDF ontology that conforms to SCROs descriptions of source-code. This way, we provide a clean separation of the explicit OWL vocabulary with its associated schema definitions represented in SCRO from the metadata encoded in RDF. For an extended description of SCRO, the process of knowledge population, and samples of our knowledge extractor subsystem, we refer the reader to our ontologies website [1].

3.1. Recommendation procedure

Let us revisit the object-instantiation task we presented in Section 1. In this task, the user would like to find a code snippet that shows how to get a handle to the `Model` object. In order to automatically compose such a code snippet, we rely on the RDF-graph representation of the framework being reused. Figure 1 shows a partial RDF-graph representation of the desired code snippet. As described in the previous section, we can obtain this representation by parsing the Jena framework.

RDF is a flexible data-representation and graph-based modeling language. Nodes in the RDF graph are instances of OWL classes that are specified in the knowledge base. Edges, however, are SCROs object properties. For example, `hasOutputType` is a functional property that represents the method's return type. `hasInputType` represents the type of a method's formal parameter and the inverse of this

[3] `http://www.w3.org/TR/rdf-primer`

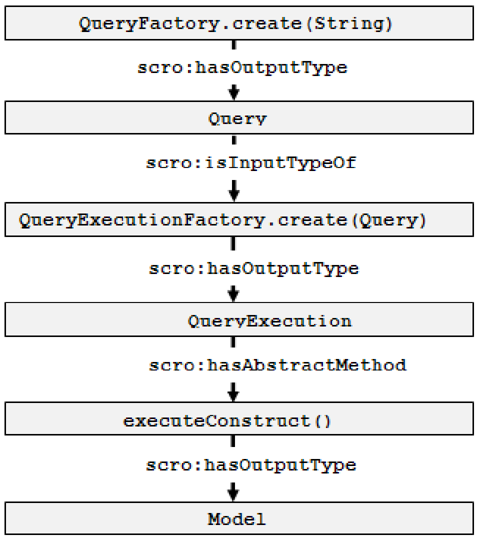

Figure 1. Answering: *Query ⇒Model*

property is `isInputTypeOf`. Inverse properties allow us to traverse the RDF graph in both directions.

Given this directed RDF graph representation, we construct a code snippet through a guided brute-force graph-traversal search starting at the node that represents the source type (`Query`), enumerating all possible path candidates to the given destination type (`Model`). However, traversing and reasoning over large graphs can be an expensive operation. In order to avoid reasoning overhead, we use the reasoner to obtain the inference closure of the original model. We then save the result which includes the computed entailments into a plain ontology model. This new model will be used for further processing. This way, we maintain the benefits of inference but avoid the added costs implied by using the reasoner. Furthermore, since we are only interested in object-instantiation queries, not every path in the graph is of interest to us. Therefore, we restrict the obtained plain model to only those RDF nodes and edges that can ever be used in a path that represents a code snippet from the given source object to the required destination object. The obtained model will be used as the basis for graph traversal, querying, and hence snippet construction. We call this model the *Snippet Model*.

3.2. Handling Downcasts

At its current state, the *Snippet Model* has no support for handling narrowing reference conversions (downcasts). We thus introduce the `hasActualOutputType` OWL property from SCRO. This property represents the actual runtime return type of methods. Therefore, we enrich the RDF graph with an edge labeled with `hasActualOutputType` that leaves a given method and enters every possible runtime return type of this method. During snippet construction, `hasActualOutputType` is treated as an expression casting the result of the method down to its actual return type.

In order to obtain the precise return type of API methods, we rely on inter-procedural points-to and call graph analysis. Points-to analysis [3] is a static program analysis technique that analyzes a sample program in order to obtain precise reference and call-target information. The essence of using this technique in our approach is to analyze each API method in order to obtain a points-to set of its possible runtime return types. Once obtained, this information is used to enrich the snippet model with paths, along the `hasActualOutputType` property, between a given method and its actual runtime type. This information will be used to determine the legality of a type cast when search is performed at a later stage. In order to perform this pre-processing step, we use the Soot framework [14]; a popular program analysis and optimization framework for Java. Soot is capable of performing inter-procedural data-flow analysis on whole-program points-to graph mode. However, this precise analysis requires creating program entry points. Although we are currently working on methods that make points-to analysis viable on programs that have no distinguished entry point, this is still an ongoing work. Therefore, we currently write appropriate entry points manually by providing a main class with a main method that exercises the API in question.

3.3. Snippet ranking and selection

Since every obtained code snippet represents a path in the graph, it is notable that answering a given query may result in a large number of paths; each of which is a solution candidate. However, not all solutions are of the same degree of relevancy to the user. It is also notable that no recommendation mechanism can rely on the type system to identify best candidate solutions. Therefore, heuristics can be used to rank the results based on relevancy measures to the task at hand. In our approach, we use the path size heuristic as well as user context heuristics.

Shortest-Path-First (SPF) is a simple yet proven effective heuristic. It assigns top rank to the shortest path in the graph. This heuristic is a variation of the code length heuristic proposed by Prospector[7] and used by others. However, in our case, a path size represents the number of RDF statements that are necessary to compose the snippet. Therefore, the size of the path in Figure 1 is five.

While SPF clusters and ranks paths based on their size, context-based heuristic assigns higher ranks to paths that better fit within the current user context. We thus analyze the code that is currently being developed by the user, then we create a context profile that include all visible types that are either declared by the user or inherited in the user's context. We further analyze each retrieved path in terms of the new types that this path will introduce into the current context. For example, a particular method invocation may have an argument that requires instantiating and thus introducing a new type into the current context. Naturally, code snippets that introduce more types should be assigned a lower rank value. However, if the newly introduced type is found in the context profile, it will not count against the enclosing path. This is entirely based on a simple scoring procedure that accounts for the number of newly introduced types and their visibility in the context profile. These heuristics are simple, easy to implement, and helped improve our results.

4. Implementation and evaluation

We have implemented RECOS, a prototype object-instantiation and recommendation system. RECOS is currently combined with a tool we have developed for detecting design pattern instances in object-oriented frameworks [2]. This combination is meant to promote multiple levels of software understanding and knowledge reuse. It is evident, based on empirical studies [6], that code examples are undoubtedly necessary for understanding framework usage, however, examples alone may not be enough to achieve the full potential of systematic software reuse. To be truly effective, developers need to learn the design knowledge implemented in these reusable frameworks. Our current implementation provides both advantages in one tool.

In order to use RECOS, one needs to provide the location of the framework's binaries. The knowledge extractor subsystem automatically parses the jar files and generates a RDF ontology representing the structural description of the framework's API. This ontology is classified by the reasoner to generate semantic entailments and ensure proper conformance to SCRO's vocabulary and constraints. Once classified, the system automatically generates the snippet model that is subsequently enriched with ontological instances obtained via points-to analysis as described in Sections 3.1 and 3.2, respectively. The final ontology serves as the basis for answering object-instantiation queries. The recommender subsystem accepts a user query (using a very simple input form for entering the source and destination objects), performs graph traversal, selects and ranks appropriate paths, and generates a custom code snippet for each path. This subsystem is independent from the knowledge generator subsystem. A user need only to configure it with the location of the snippet ontology, the directory of the code currently being edited, and the number of code snippets returned when a query is executed (default is 15).

In order to assess the benefits acquired by our approach, we conducted multiple experiments. The fundamental guiding hypotheses we test in these experiments are:

H 1 *Ontology-based representation of source-code knowledge improves search precision.*

H 2 *Inter-procedural points-to analysis techniques relieve a recommendation system from relying on a repository populated with sample client code.*

H 3 *Contextual information provides better ranking and filtering of the recommended items.*

4.1. Case study: framework usage

This experiment is designed to evaluate RECOS accuracy for answering object-instantiation tasks. We have selected ten Jena programming tasks. These tasks vary in their complexity; ranging from a simple constructor or static method call to a complex sequence of expressions. Table 1 shows statistics about these tasks after being expressed as object-instantiation queries.

Table 1. RECOS framework usage results

Task	*Source*	*Destination*	Context	Rk.[1]
T1	Statement	RDFDataType	-	1
T2	RDFList	IntersectionClass	-	1
T3	Resource	ComplementClass	-	3
T4	*Null*	OntModel	String	1
T5	ResultBinding	RDFNode	-	1
T6	Statement	RDFList	Property, OntClass	2
T7	IDBConnection	ModelRDB	-	1
T8	String	Individual	OntModel, String	0
T9	Query	Model	String, OntModel	3
T10	Resource	OntModel	Model	3

[1] Rk: Rank of desired solution if found, 0 otherwise.

For each task, an environment has been setup such that the desired code snippet is left incomplete. We then instructed RECOS to fill in the missing code. We remind the reader that a desired solution to a given query may not be completely ready for immediate insertion in user code. In some cases, the user still need to issue another query to instantiate one or more objects introduced by the solution (e.g., an argument in a method call or an intermediate object within the sequence). Consider task T3 for example, the retrieved code snippet requires an object of type `OntModel` to be present for the code to compile. RECOS generates an intermediate variable that suggests a need to instantiate this object. Typically, these objects can be instantiated with a query specifying only the destination object as seen in task T4. Usually, generalized queries of this form produce many hits. However, precise ontology-based representations of API code combined with heuristics produce a good rank of the desired result. This shows a clear support for hypotheses H1 and H2. Hypothesis H2 is also supported in part by tasks T5-T7. In task T5 for example, an intermediate method that returns an object of type `Object` needed to be converted to `RDFNode`. Points-to analysis inferred that this cast is possible, thus, avoided long and undesired paths.

In some cases, it was extremely difficult to infer the user's intent. Consider task T8 for example, RECOS returns many hits that do not complete the task in question. This task in fact shed a light on one of the difficulties faced by recommendation systems in general. Dealing with String objects usually affects precision and requires more sophisticated approaches to be handled properly. Furthermore, API methods that are heavily overloaded have their own affects on ranking. In task T9, RECOS ranked the desired solution third and generates many hits. As described in Section 3.1, this snippet requires invoking a heavily overloaded method that creates a `QueryExecution` to execute over the ontology model. However, context heuristics filtered out plenty of paths that would otherwise get a higher rank. Hypothesis H3 was also supported by T10. RECOS, in this case, filtered out many paths that would have introduced new objects (e.g., `OntModelSpec`) into the user context.

4.2. Comparison with other approaches

It is not trivial to compare code search tools due to the obvious lack of standard benchmarks and tool availability issues. We thus extend a case study proposed by Thummalapenta and Xie [13] and used to evaluate PARSEWeb against Prospector and Strathcona. This case study is based on the Logic example project of the Eclipse Graphical Editing Framework (GEF)[4]. The authors proposed ten programming tasks that are shown in Table 2. We parsed the needed jar files, generated the GEF Snippet Model, and instructed RECOS to find solutions for each task.

Table 2. GEF Logic tasks & results

Task	*Source*	*Destination*	Rank
T1	IPageSite	IActionBars	1
T2	ActionRegistry	IAction	1
T3	ActionRegistry	ContextMenuProvider	1
T4	IPageSite	ISelectionProvider	1
T5	IPageSite	IToolBarManager	1
T6	String	ImageDescriptor	1
T7	Composite	Control	10
T8	Composite	Canvas	7
T9	GraphicalViewerThumbnail	Scrollable	0
T10	GraphicalViewer	IFiugure	0

As observed in Table 2, RECOS found solutions for all tasks except T9 and T10. T9 was in fact infeasible since we could not verify the existence of the source object in the library. T10 was not answered by any of the tools. In fact, PARSEWeb was unable to find a solution

[4] `http://www.eclipse.org/gef/`

for T3; Prospector and Strathcona could not answer T6-T8. Consider T8 for example, the shortest solution would be to invoke an existing `Canvas` constructor that accepts `Composite` in its parameter list. However, a desired solution based on the user's context is, in fact, to instantiate `PageBook`, a sub-type of `Composite`, and pass that object to the constructor. Without our context heuristics, RECOS would have ranked this answer further down in the list. PARSEWeb ranked this higher since it utilizes the usage frequency heuristic. Prospector, on the other hand, could not answer the query or perhaps the answer did not show up in the list[5] because its ranking mechanism is based mostly on the length heuristic.

These results show a clear support for our three hypotheses. However, PARSEWeb outperforms RECOS and Prospector [6] in the number of retrieved results. This is expected due to our reliance on API structures, but the effect of the final results was greatly reduced due to the nature of the semantic representation and organization of API knowledge. Furthermore, PARSEWeb ranked some of the desired solutions higher. PARSEWeb performs sequence post-processing and clustering that appears to improve the total number of retrieved results and plays a role in ranking similar sequences. However, PARSEWeb relies on incomplete code fragments obtained from GCS and has no access to API information. Therefore, it must use various heuristics for type resolution. These heuristics, however, may not work when the downloaded code contains a complex sequence of method calls that was not used in initialization expressions. Achieving perfection in code search is near impossible, however, we believe that RECOS internal mechanisms proved effective; and in the majority of cases, show a clear support for our three hypotheses.

5. Discussion and future work

Ontologies have been widely recognized as effective means for knowledge representation. On the other hand, points-to analysis techniques provide effective mechanisms for type inference. In this paper, we proposed an approach that combines the strengths of both techniques to improve search for relevant source-code snippets. We have also developed RECOS, a code search tool for object-instantiation specific queries. RECOS is currently not tied to a particular IDE. However, we are currently integrating RECOS into Eclipse as part of a comprehensive tool for program understanding and knowledge reuse. In addition to snippet recommendation and design recovery, this tool will be used to recommend reusable components (e.g., finding source and/or destination objects). We are also investigating the application of semantic annotations and domain ontologies to improving search precision and ranking.

[5]Prospector was configured to show only the first 12 results

[6]Strathcona did not perform well in this experiment since it does not have a clear support for object-instantiation queries

References

[1] A. Alnusair and T. Zhao. Source Code Ontology (SCRO) and examples of automatic ontology population. `http://www.cs.uwm.edu/~alnusair/ontologies`.

[2] A. Alnusair and T. Zhao. Towards a model-driven approach for reverse engineering design patterns. In *Proc. 2nd International Workshop on Transforming and Weaving Ontologies in Model Driven Engineering (TWOMDE'09)*, 2009.

[3] M. Emami, G. Rakesh, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. Conference on Programming Language Design and Implementation (PLDI)*, pages 242–256, 1994.

[4] T. R. Gruber. A translation approach to portable ontology specification. *Knowledge Acquisition*, 5(2):192–220, 1993.

[5] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proc. International Conference on Software Engineering (ICSE)*, pages 117–125, 2005.

[6] D. Hou. Investigating the effects of framework design knowledge in example-based framework learning. In *Proc. IEEE International Conference on Software Maintenance*, pages 37–46, 2008.

[7] D. Mandelin, L. Xu, L. Bodik, and D. Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. ACM Conference on Programming Language Design and Implementation*, pages 48–61, 2005.

[8] A. Michail. Data mining library reuse patterns using generalized association rules. In *Proc. International Conference on Software Engineering (ICSE)*, pages 167–176, 2000.

[9] D. Poshyvanyk, A. Marcus, and Y. Dong. JIRiSS-an eclipse plug-in for source code exploration. In *Proc. IEEE Conference on Program Comprehension*, pages 252–255, 2006.

[10] M. P. Robillard, R. J. Walker, and T. Zimmermann. Recommendation systems for software engineering. *IEEE Software*, 27(4):80–86, 2010.

[11] R. Sindhgatta. Using an information retrieval system to retrieve source code samples. In *Proc. International Conference on Software Engineering*, pages 905–908, 2006.

[12] N. Tansalarak and K. Claypool. XSnippet: mining for sample code. In *Proc. ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 413–430, 2006.

[13] S. Thummalapenta and T. Xie. PARSEWeb: a programmer assistant for reusing open source code on the web. In *Proc. International Conference on Automated Software Engineering*, pages 204–213, 2007.

[14] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a Java bytecode optimization framework. In *Conference of the Centre for Advanced Studies on Collaborative Research*, pages 242–256, 1999.

[15] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. MAPO: Mining and recommending api usage patterns. In *Proc. European Conference on Object Oriented Programming (ECOOP'09)*, pages 318–343, 2009.